Licenciatura Engenharia Informática e Multimédia

Computação Física – CF

Relatório Trabalho Prático 2

Docente Carlos Carvalho

Trabalho realizado por:

Fábio Dias, nº 42921

Tatiana Cristão, nº 47508

Índice

Índice de Figuras

Índice de Tabelas

1. Desenho do Microprocessador

Para este trabalho prático, foi-nos pedido para desenhar um microprocessador, baseando-nos na arquitetura Harvard e simulá-lo no Arduino.

A arquitetura Harvard é composta pela Memória de Código, Memória de Dados e pelo CPU. Ambas as memórias possuem um *Adress Bus* e um *Data Bus*. O *Data Bus* da Memória de Código deve ser apenas de leitura e o *Adress Bus* deve ser apenas de escrita. O *Adress Bus* da Memória de Dados também deve ser apenas de escrita, mas o *Data Bus* da Memória de Dados pode ser de escrita e de leitura, mas apenas um destes estados está ativo. *(Ver Figura 1)*.

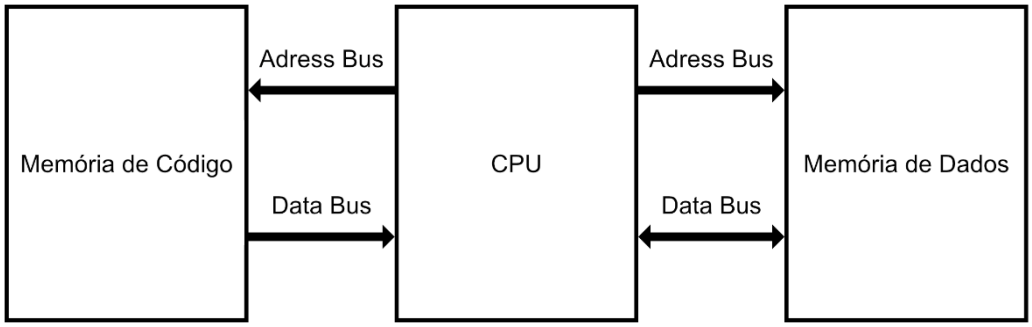

Figura 1 - Arquitetura Harvard

A Memória de Código é onde o programa é cumprido. Este é apenas de leitura exceto quando queremos outro programa. Nesse caso, a memória tem de ser regravada.

A Memória de Dados possui as variáveis do programa.

O CPU, *Central Processing Unit*, é onde o programa é cumprido.

A partir do enunciado, conseguimos especificar o Registo *A*, um conjunto de Registos *Rn*, constituído por dois registos, *R0* e *R1*, o Registo de Controlo de Execução, *PC*, as *flags Carry* e *Borrow*, *Cy*, *Overflow*, *Ov*, e *Zero*, *Z*.

Analisando as instruções da tabela do enunciado, conseguimos concluir que o Registo *PC*, na última instrução, toma um valor de sete bits. Esta é a dimensão máxima que este registo pode ter, dado que as outras instruções de controlo de fluxo são a seis bits. Logo, o Registo *PC* tem uma dimensão de sete bits, que corresponde ao *Adress Bus* da Memória de Código. Na primeira instrução movemos para um dos Registos *R*, o valor de uma constante de cinco bits, logo, ambos os Registos *R* têm uma dimensão de cinco bits que corresponde ao *Adress Bus* da Memória de Dados. Isto porque o *Adress Bus* da Memória de Dados tem a mesma dimensão do registo que endereça a memória que, pelas instruções do enunciado, é um dos Registos *Rn*. Na segunda instrução, movemos para o Registo *A* o valor de um dos Registos *R*. Logo, também tem uma dimensão de cinco bits e este corresponde ao *Data Bus* da Memória de Dados. Para obtermos a dimensão do *Data Bus* da Memória de Código é necessário codificarmos cada uma das instruções, de forma a distingui-las e incluirmos todos os seus parâmetros.

Seguindo para a codificação das instruções, como referido previamente, o objetivo é conseguimos distinguir todas as instruções, incluindo os parâmetros das instruções em causa. Conseguimos atingir este objetivo com dez bits, dos quais, quatro são de distinção. Estes são o D9 D8, D1 e D0. Assim concluímos que o Data Bus da Memória de Código tem uma dimensão de dez bits. *(Ver Tabela 1)*

Instruções	Parâmetros	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
MOV Rn ,const5	Rn, const5	0	0	Rn	c4	c3	c2	c1	c0	0	0
MOV A ,Rn	Rn	0	0	Rn	0	0	0	0	0	0	1
MOV Rn , A	Rn	0	0	Rn	0	0	0	0	0	1	0
MOV A , @Rn	Rn	0	0	Rn	0	0	0	0	0	1	1
MOV @Rn , A	Rn	0	1	Rn	0	0	0	0	0	0	0
CPLF	----------	0	1	0	0	0	0	0	0	0	1
NOT A	----------	0	1	0	0	0	0	0	0	1	0
AND A , Rn	Rn	0	1	Rn	0	0	0	0	0	1	1
OR A , Rn	Rn	1	0	Rn	0	0	0	0	0	0	0
ADDC A , Rn	Rn	1	0	Rn	0	0	0	0	0	0	1
SUBB A , Rn	Rn	1	0	Rn	0	0	0	0	0	1	0
JC rel6	rel6	1	1	r5	r4	r3	r2	r1	r0	0	0
JNZ rel6	rel6	1	1	r5	r4	r3	r2	r1	r0	0	1
JOV rel6	rel6	1	1	r5	r4	r3	r2	r1	r0	1	0
JMP end7	end7	1	e6	e5	e4	e3	e2	e1	e0	1	1

Tabela 1 - Codificação das Instruções

Para efetuarmos esta tabela, simplesmente identificamos os parâmetros, caso existam, e colocamos a sua dimensão de forma modular, como é possível identificar nas três linhas finais, por exemplo. De seguida identificamos os bits de distinção e tentamos preencher a tabela até chegarmos a valores únicos, ou seja, não repetidos. Portanto, como podemos observar na tabela, não existe nenhuma combinação repetida com os quatro bits de distinção.

De seguida partimos para o desenho do módulo funcional. *(Ver Figura 2)*

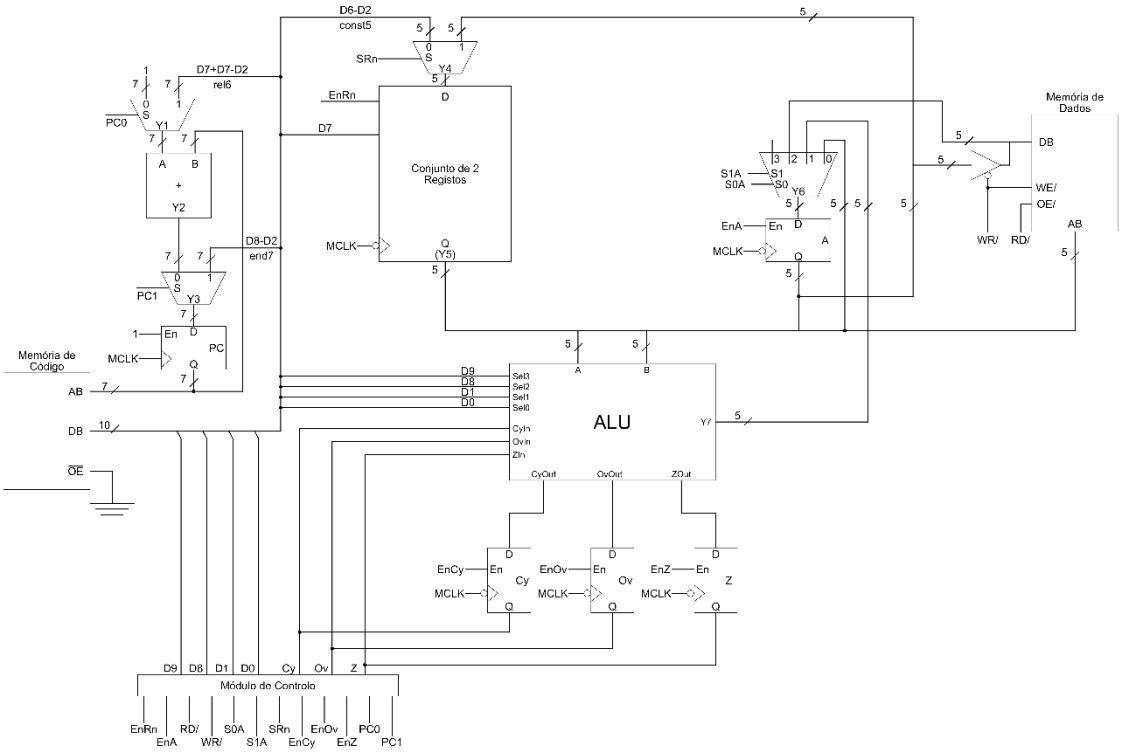

Figura 2 - Módulo Funcional

Começando pelo Registo de Controlo de Execução, o Registo *PC*, este encontra-se a 0. No bloco somatório, vai receber o valor presente, ou seja, 0, e um valor que provém do multiplexer acima, *Y1*. Esta saída toma um valor entre 1 ou um número relativo proveniente do *Data Bus* que é um número relativo, a 6 bits, mas, dado que *PC* é a sete, este valor necessita que o sinal seja estendido, repetindo o bit de maior peso. Este *Y1* depende do sinal de saída do Módulo de Controlo *PC0*. Após a soma, é produzido um valor para a saída *Y2*. Esta entra no multiplexer abaixo que decide, a partir do seletor *PC1*, se o registo *PC* vai guardar o *Y2* ou o valor que vem da Memória de Código a partir dos sete bits, D8 a D2. *(Ver Figura 3)*

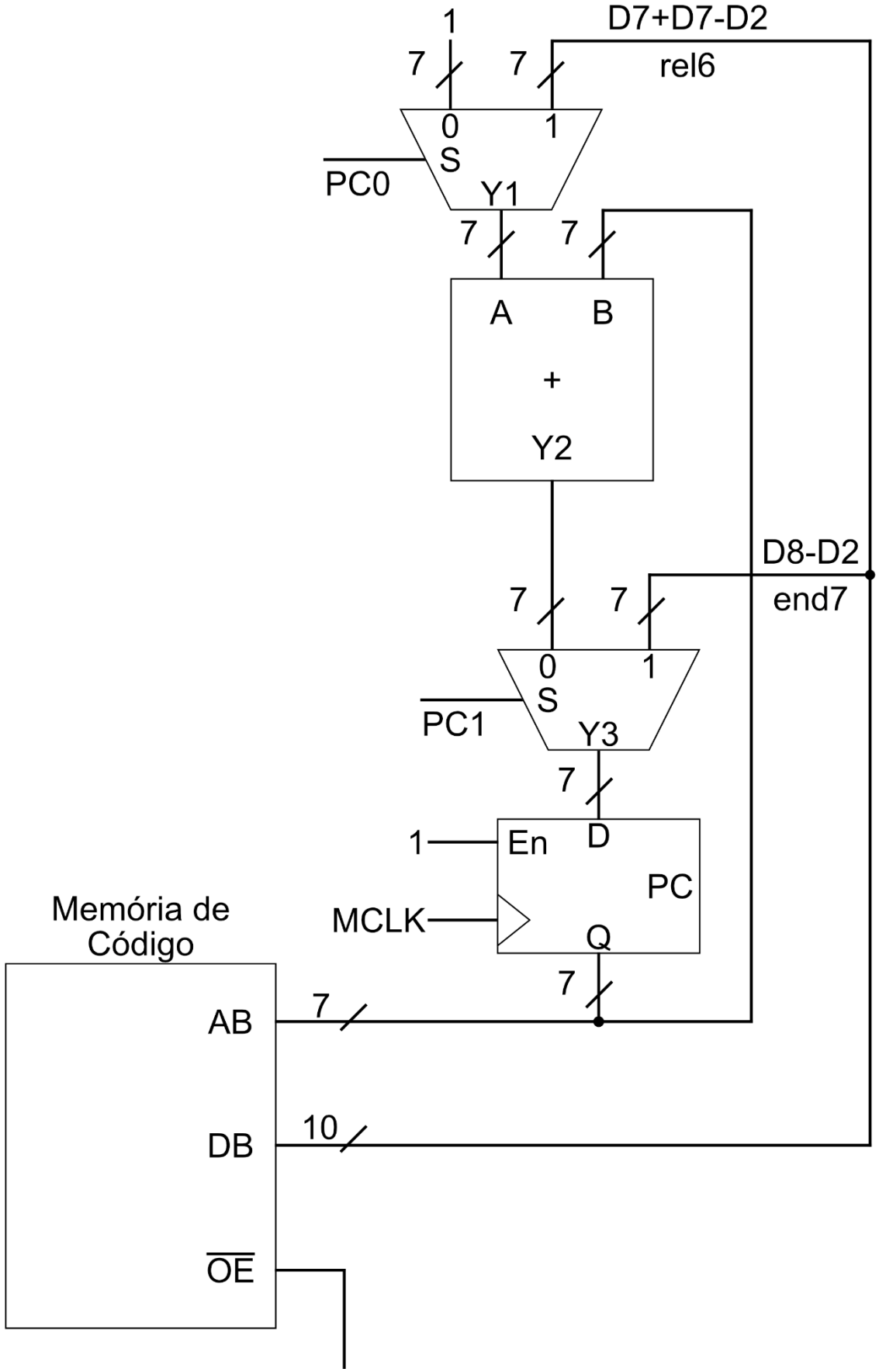

Figura 3 - Registo PC

Para os Conjunto de Registos *R*, o valor recebido depende do que provém do *Data Bus* da Memória de Código, mais especificamente dos bits D6, D5, D4, D3, D2, ou do que se encontra no Registo *A*. Esta escolha é feita a partir de um multiplexer, cujo seletor é o sinal de saída do Módulo de Controlo, *SRn*, que produz a saída *Y4*. Esta é dirigida para as entradas de cada Registo *R*. O *enable* de cada um destes é definido pelo demultiplexer, cujo seletor é o bit D7 proveniente do *Data Bus* da Memória de Código. Por fim, ambas as saídas são entradas do multiplexer, cujo seletor também é o bit D7, e que produz a saída *Y5*. *(Ver Figura 4)*

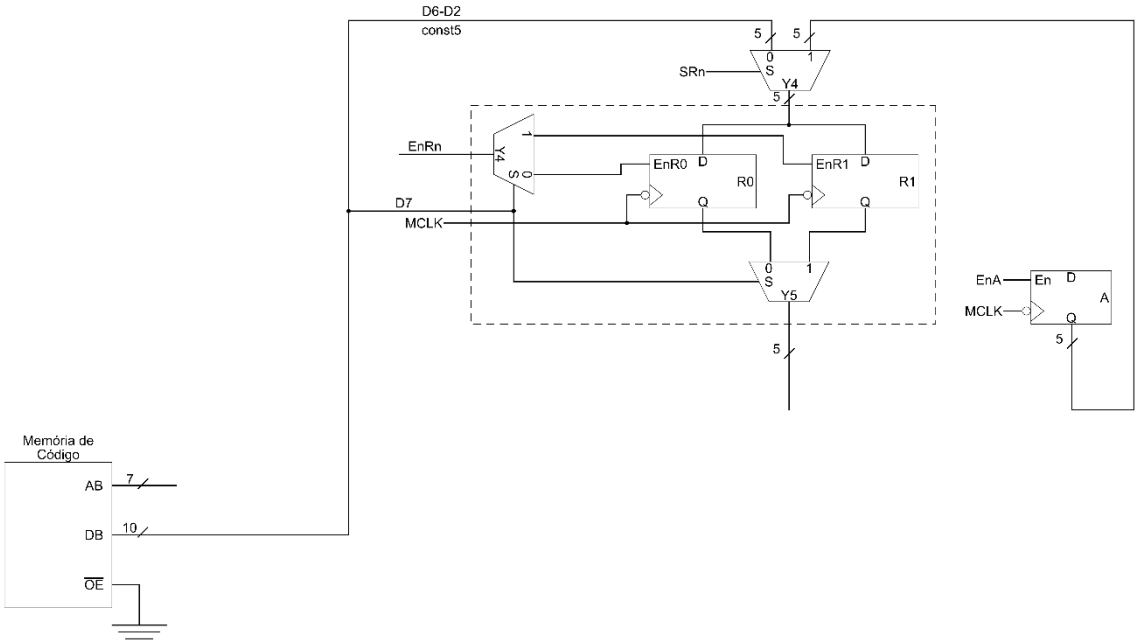

Figura 4 - Registo Rn

No caso do Registo *A*, a sua entrada é proveniente de um multiplexer 4 por 2, cuja saída é *Y6*. Não há nenhum valor associado à quarta entrada do multiplexer, mas a terceira é proveniente do *Data Bus* da Memória de Código. A segunda entrada do multiplexer possui o resultado da *ALU*, *Arithemtic Logic Unit*, que será explicada mais à frente. Por fim, a primeira entrada do multiplexer é proveniente da saída *Y5*, que é o valor que um dos Registos *R* possui. Este multiplexer é controlado pelos sinais de saída do Módulo de Controlo, *S0A* e *S1A*. *(Ver Figura 5)*

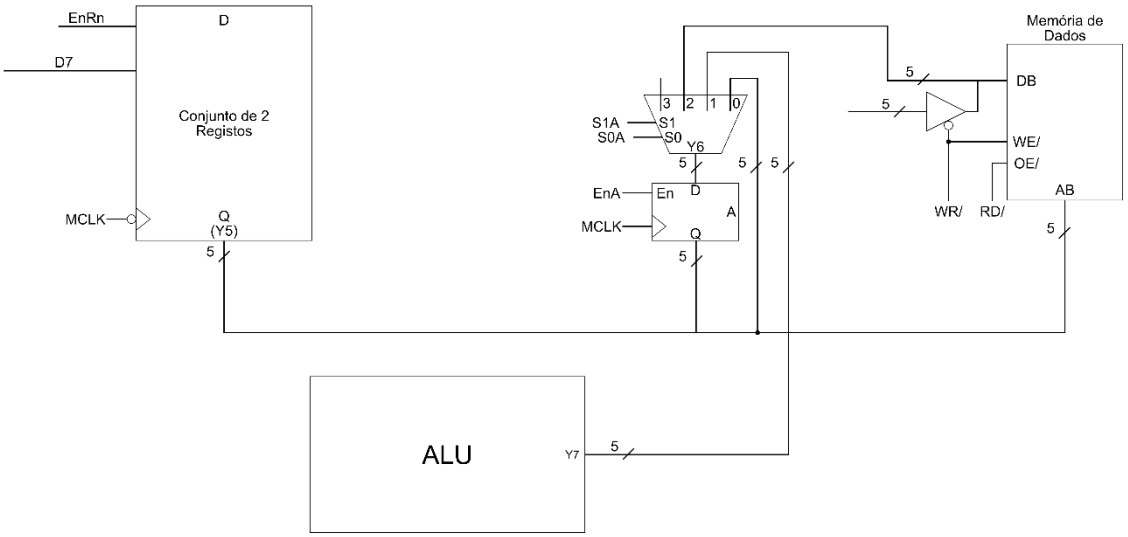

Figura 5 - Registo A

Com mencionado, existe uma *ALU* e está é constituída por duas entradas, *A* e *B*, que produz quatro saídas: A *Y7*, e as outras três são as *flags CyOut*, *OvOut* e *ZOut*. Estão são guardadas nos respetivos registos, com os *enable*'s dependentes dos sinais de saída do Módulo de Controlo, *EnCy, EnOv* e *EnZ*. As saídas destes registos também são entradas da *ALU* sob a forma de *CyIn*, *OvIn* e *ZIn*. De forma a selecionar a operação pretendida, esta é distinguida pelos bits D0, D1, D8 e D9 provenientes do *Data Bus* da Memória de Código. Apesar de termos apenas seis operações, devido à forma como as instruções forma codificadas, vão ser precisos estes quatro bits para as separar a todas. *(Ver Figura 6)*

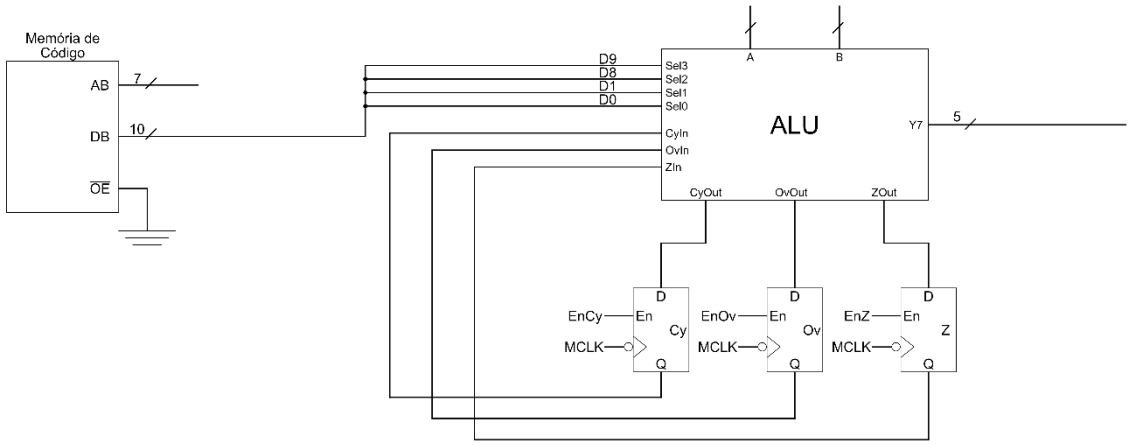

Figura 6 - ALU, Arithmetic Logic Unit

Por fim, temos a escrita e leitura do *Data Bus* da Memória de Dados. Dado que este se comporta como uma entrada assim como uma saída, temos de garantir que não pode ter os dois comportamentos ao mesmo tempo. Desta forma, usamos um *tri-state* que, a partir do sinal de saída do Módulo de Controlo, *WR/*, possuímos ou não, alta impedância, como se fosse uma resistência infinita. Neste caso, o seu e*nable* é ativo a 0, daí o sinal ser complementado. *(Ver Figura 7)*

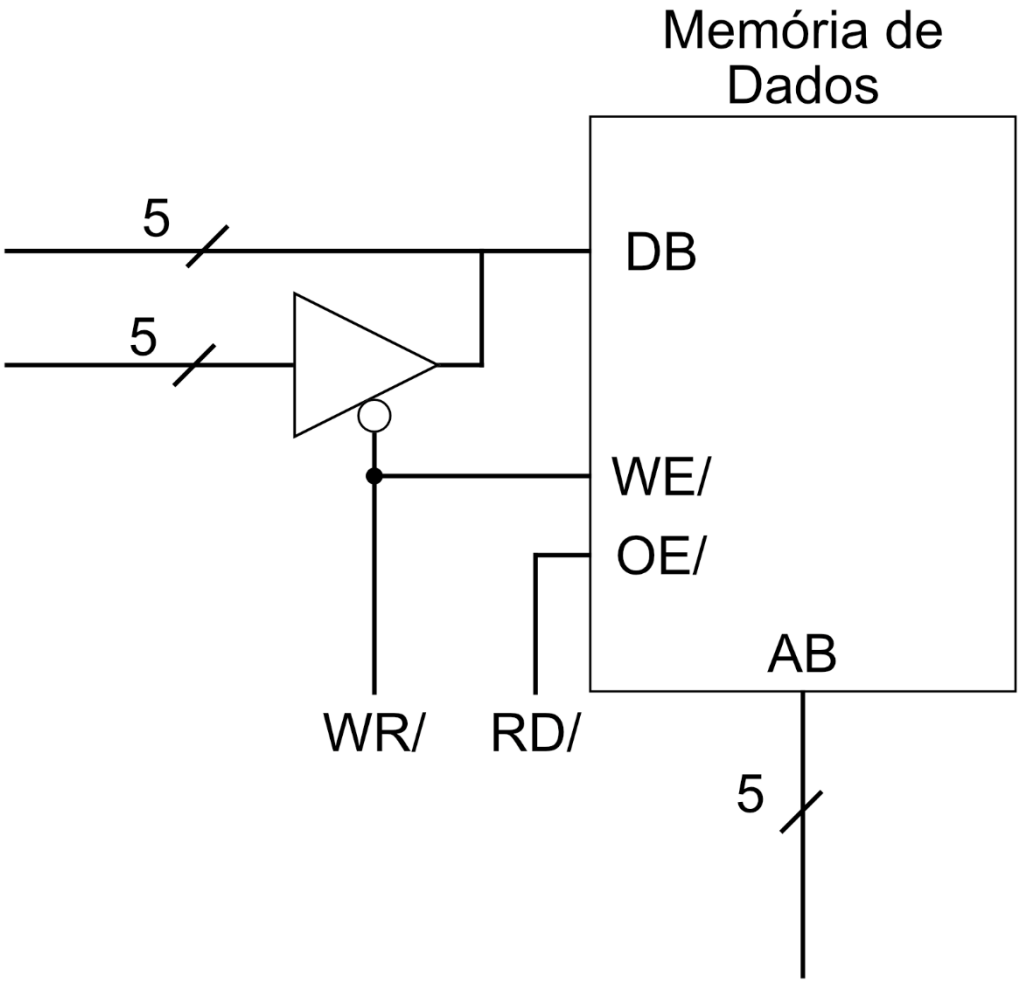

Figura 7 - Tri-State

Foram apresentados vários sinais de saída do Módulo de Controlo. Mas a realidade é que o sinal *PC0* é constituído por três diversas causas. Ocorre quando a *flag Cy* é 1, quando a *flag Ov* é 1 e quando a *flag Z* é 0. Ou seja, podemos concluir que cada uma destas causas tem de ter um sinal próprio e é o conjunto destes sinais que produz o valor de *PC0*. Ou seja, uma expressão AND-OR com um *AND* entre *JC* e *Cy*, um *AND* entre *JOv* e *Ov* e um *AND* entre *JNZ* e *Z/*. O sinal *PC1* depende apenas da última instrução, logo, chamemos-lhe *JMP (Ver Figura 8)*

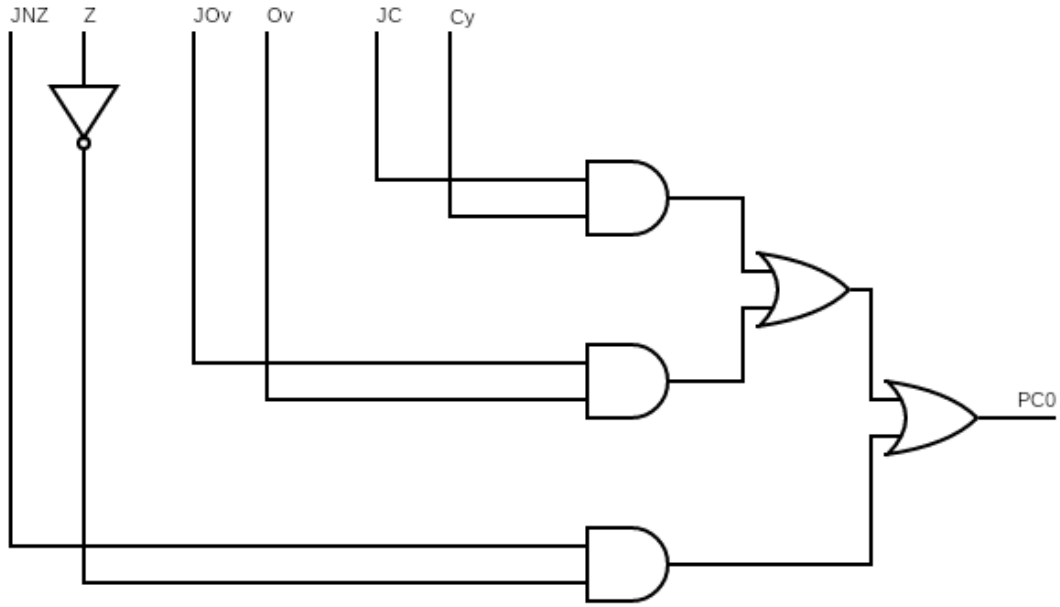

Figura 8 - Circuito PC0

Como é possível observar, o CLK do Registo *PC* é diferente de todos os outros Registos. Isto porque a transição ascendente do CLK atualiza o Registo *PC* enquanto a transição descendente do CLK atualiza todos os outros. Isto provém do facto de ter de existir um tempo físico entre a atualização do Registo *PC* e dos valores combinatórios a serem registados devido à instrução atual ter sido atualizada. O tempo entre a transição ascendente e a descendente é suficiente para isso.

Observando cada instrução no módulo funcional, podemos obter uma especificação das entradas do módulo de controlo, assim como as suas saídas. Assim, criamos a tabela de verdade do Módulo de Controlo. Se juntarmos o valor de todos os sinais de saída de cada instrução e o convertemos para um valor hexadecimal, conseguimos obter um código. Este será usado posteriormente. *(Ver Tabela 2)*

D9	D8	D1	D0	Sinais Activos	EnRn	EnA	RD/	WR/	S0A	S1A	SRn	EnCy	EnOv	EnZ	JC	JNZ	JOv	JMP	Hex
0	0	0	0	EnRn	1	0	1	1	0	0	0	0	0	0	0	0	0	0	2C00
0	0	0	1	EnA	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1C00
0	0	1	0	EnRn, SRn	1	0	1	1	0	0	1	0	0	0	0	0	0	0	2C80
0	0	1	1	EnA, RD/, S1A	0	1	0	1	0	1	0	0	0	0	0	0	0	0	1500
0	1	0	0	WR/	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0800
0	1	0	1	EnCy, EnOv, EnZ	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0C70
0	1	1	0	EnA, S0A, EnZ	0	1	1	1	1	0	0	0	0	1	0	0	0	0	1E10
0	1	1	1	EnA, S0A, EnZ	0	1	1	1	1	0	0	0	0	1	0	0	0	0	1E10
1	0	0	0	EnA, S0A, EnZ	0	1	1	1	1	0	0	0	0	1	0	0	0	0	1E10
1	0	0	1	EnA, S0A, EnCy, EnOv, EnZ	0	1	1	1	1	0	0	1	1	1	0	0	0	0	1E70
1	0	1	0	EnA, S0A, EnCy, EnOv, EnZ	0	1	1	1	1	0	0	1	1	1	0	0	0	0	1E70
1	1	0	0	JC	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0C08
1	1	0	1	JNZ	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0C04
1	1	1	0	JOv	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0C02
1	x	1	1	JMP	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0C01

Tabela 2 - Módulo de Controlo

Finalmente, adicionamos a ROM e definimos a gama de endereços para cada instrução. Convertemos o valor dos bits D9, D8, D1 e D0, obtendo um código hexadecimal. Este será o índice que armazena o código formado anteriormente dos sinais de saída do Módulo de Controlo. *(Ver Tabela 3)*

D9	D8	D1	D0	Gama de Endereços	EnRn	EnA	RD/	WR/	S0A	S1A	SRn	EnCy	EnOv	EnZ	JC	JNZ	JOv	JMP	Hex
0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	2C00
0	0	0	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1C00
0	0	1	0	2	1	0	1	1	0	0	1	0	0	0	0	0	0	0	2C80
0	0	1	1	3	0	1	0	1	0	1	0	0	0	0	0	0	0	0	1500
0	1	0	0	4	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0800
0	1	0	1	5	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0C70
0	1	1	0	6	0	1	1	1	1	0	0	0	0	1	0	0	0	0	1E10
0	1	1	1	7	0	1	1	1	1	0	0	0	0	1	0	0	0	0	1E10
1	0	0	0	8	0	1	1	1	1	0	0	0	0	1	0	0	0	0	1E10
1	0	0	1	9	0	1	1	1	1	0	0	1	1	1	0	0	0	0	1E70
1	0	1	0	A	0	1	1	1	1	0	0	1	1	1	0	0	0	0	1E70
1	1	0	0	C	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0C08
1	1	0	1	D	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0C04
1	1	1	0	E	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0C02
1	x	1	1	B, F	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0C01

Tabela 3 - Módulo de Controlo com ROM

Possuindo todas estas especificações, conseguimos simular o microprocessador no Arduíno.

2. Implementação do Microprocessador

Dado que o CLK é manual, com auxílio a um botão, definimos o pino do botão e, como foi feito no trabalho anterior, definimos duas variáveis de controlo de tempo para evitar o *debounce*. Este será definido como *INPUT_PULLUP*. *(VerFigura 9)*

```
//Definir botão para o CLK
#define pinCLK 0

//Variáveis de Controlo de Tempo do Botão
unsigned long now, ago;
```

Figura 9 - Código Variáveis Botão

Com base no desenho do microprocessador, estabelecemos que possuímos uma memória de Código, *ROM*, e uma Memória de Dados, *RAM*, sendo estas simuladas a partir de *array*'s. Possuem, respectivamente, cento e vinte e oito e trinta e dois índices. *(Ver Figura 10)*

```
//Memória de Código
int ROM[128];

//Memória de Dados
byte RAM[32];
```

Figura 10 - Código Variáveis ROM e RAM

Teremos também as saídas do Módulo de Controlo, num *array* de dezasseis bits, designado por *ROM_MC*. Este é inicializado e preenchido manualmente. (*Ver Figura 11)*

```
//Definições da ROM_MC
const int ROM_MC[16] = {0x2C00, 0x1C00, 0x2C80, 0x1500, 0x0800, 0x0C70, 0x1E10, 0x1E10, 0x1E10, 0x1E70, 0x0C01, 0x1E70, 0x0C08, 0x0C04, 0x0C02, 0x0C01};
```

Figura 11 - Código Variáveis ROM_MC

Adicionalmente, definimos os registos. Temos de definir duas variáveis para cada um. Uma entrada, *D*, e uma saída, *Q*, que só é afetada quando existir *enable*. Como são apenas afetadas na transição ascendente ou descendente do CLK, as saídas têm de ser *volatile*. *(Ver Figura 12)*

```
//Entradas e Saídas dos Registos
byte DPC;               //Entrada Registo PC
volatile byte QPC;      //Saída Registo PC
byte DRn[2];            //Entrada Registo Rn
volatile byte QRn[2];   //Saída Registo Rn
byte DA;                //Entrada Registo A
volatile byte QA;       //Saída Registo A
boolean DCy;            //Entrada Flag Carry
volatile boolean QCy;   //Saída Flag Carry
boolean DOv;            //Entrada Flag Overflow
volatile boolean QOv;   //Saída Flag Overflow
boolean DZ;             //Entrada Flag Zero
volatile boolean QZ;    //Saída Flag Zero
```

Figura 12 - Código Variáveis Registos

De seguida, definimos as variáveis que correspondem às saídas do Módulo de Controlo. Estas só possuem um bit, logo serão *boolean*'s. *(Ver Figura 13)*

```
//Saídas do Módulo de Controlo
boolean EnRn, EnA, RD, WR, S0A, S1A, SRn, EnCy, EnOv, EnZ, JC, JNZ, JOv, JMP;
```

Figura 13 - Código Variáveis Saídas Módulo de Controlo

Como foi apresentado previamente no Módulo Funcional, existem saídas designadas por *Y*. No código, também as vamos definir de forma a facilitar a organização do módulo combinatório. *(Ver Figura 14)*

```
//Variáveis de Saída
byte Y1, Y2, Y3, Y4, Y5, Y6, Y7;
```

Figura 14 - Código Variáveis de Saída

Na implementação, foi possível observar que iriamos ter de obter o conteúdo da Memória de Código da instrução atual. De forma a facilitar o seu acesso, foi definida uma variável global que é afetada com essa instrução. *(Ver Figura 15)*

```
//Código
int codigo;
```

Figura 15 - Código Variáveis Código

Começaremos por definir todas as estruturas que vamos precisar, com base no desenho do Módulo Funcional. Vamos precisar de Multiplexer's, tanto dois para um como quatro para dois, Registos com Enable, para *bytes* e para *boolean*'s, este designado por flip-flop-D, e também um Demultiplexer. *(Ver Figuras 16, 17, 18, 19, 20)*

```
//MUX_2_1
void MUX_2x1(boolean sel, byte in_0, byte in_1, byte *y)
{
  *y = sel ? in_1 : in_0;
}
```

Figura 16 - Código Multiplexer 2x1

```
//MUX_4_1
void MUX_4x1(boolean sel_0, boolean sel_1, byte in_0, byte in_1, byte in_2, byte in_3, byte *y)
{
  switch(sel_1 << 1 | sel_0)
  {
    case B00:
        *y = in_0;

      break;

    case B01:
        *y = in_1;

      break;

    case B10:
        *y = in_2;

      break;

    case B11:
        *y = in_3;

      break;
  }
}
```

Figura 17 - Código Multiplexer 4x1

```
void registoComEnable(boolean enable, byte D, byte *Q)
{
  if(enable)
  {
    *Q =  D;
  }
}
```

Figura 18 - Código Registo Com Enable

```
//Flip Flop D
void flip_flop_D_Enable(boolean enable, boolean D, boolean *Q)
{
  if(enable)
  {
    *Q = D;
  }
}
```

Figura 19 - Código Flip Flop D

```
//DMUX_1_2
byte DMUX_1x2(boolean sel, boolean D)
{
  return D << sel;
}
```

Figura 20 - Código Demultiplexer 1x2

Vamos depender de duas transições do CLK, da ascendente e da descendente. Desta forma, foram criados dois métodos. De forma a manter coerência, quando um dos CLK's é invocado, via uma interrupção, esta interrupção é redefinida para o CLK contrário e o seu modo também. *(Ver Figuras 21 e 22)*

```
//Função MCLK (Ascendente)
void MCLK()
{
  now = millis();

  if(now - ago >= 200)
  {
    //Atualizar o registo PC
    registoComEnable(1, Y3, &QPC);

    attachInterrupt(pinCLK, MCLKNeg, FALLING);

    ago = now;
  }
}
```

Figura 21 - Código Master CLOCK

```
//Função MCLK Negado (Descendente)
void MCLKNeg()
{
  now = millis();

  if(now - ago >= 200)
  {
    //Atualizar os registos Rn e A, assim como as flags Cy, Ov e Z
    registoComEnable(EnRn, Y4, &QRn[bitRead(codigo, 7)]);
    registoComEnable(EnA, Y6, &QA);
    flip_flop_D_Enable(EnCy, DCy, &QCy);
    flip_flop_D_Enable(EnOv, DOv, &QOv);
    flip_flop_D_Enable(EnZ, DZ, &QZ);

    attachInterrupt(pinCLK, MCLK, RISING);

    ago = now;
  }
}
```

Figura 22 - Código Master CLOCK Negado

Passamos para o método de *AfetarSinais*, esta pega no código actual da Memória de Código, isola e agrega os bits de distinção, que servirá de índice para os sinais de saída do Módulo de Controlo, *ROM_MC*, afeta cada um dos sinais. *(Ver Figura 23)*

```
void AfetarSinais()
{
  codigo = ROM[QPC];
  byte bitsDeInstrucao = bitRead(codigo, 9) << 3 | bitRead(codigo, 8) << 2 | bitRead(codigo, 1) << 1 | bitRead(codigo, 0);
  int sinais = ROM_MC[bitsDeInstrucao];

  JMP = bitRead(sinais, 0);
  JOv = bitRead(sinais, 1);
  JNZ = bitRead(sinais, 2);
  JC = bitRead(sinais, 3);
  EnZ = bitRead(sinais, 4);
  EnOv = bitRead(sinais, 5);
  EnCy = bitRead(sinais, 6);
  SRn = bitRead(sinais, 7);
  S1A = bitRead(sinais, 8);
  S0A = bitRead(sinais, 9);
  WR = bitRead(sinais, 10);
  RD = bitRead(sinais, 11);
  EnA = bitRead(sinais, 12);
  EnRn = bitRead(sinais, 13);
}
```

Figura 23 - Código Afetar Sinais

Temos também o método simulador do Bloco de Registos, *BlocoDeRegistos*. Obtemos o valor do selector do registo, pelo sétimo bit da instrução actual, obtemos os valores dos *enable*'s dos registos, ER0 e ER1 e afetamos os registos e, de seguida, com passamos ambos os valores por um MUX_2x1, cujo selector é o sétimo bit da instrução actual. *(Ver Figura 24)*

```
//Bloco de Registos
void BlocoDeRegistos()
{
  boolean sel = bitRead(codigo, 7);
  byte enableRegistos = DMUX_1x2(sel, EnRn);
  registoComEnable(bitRead(enableRegistos, 0), Y5, &DRn[0]);
  registoComEnable(bitRead(enableRegistos, 1), Y5, &DRn[1]);
  MUX_2x1(sel, QRn[0], QRn[1], &Y5);
}
```

Figura 24 - Código Bloco de Registos

De seguida, implementá-los o bloco do ALU, que, devido à nossa codificação, recebe quatro selectores, *S_0*, *S_1*, *S_2* e *S_3*, recebe dois bytes, *A* e *B*, recebe os valores das *flags*, *CyIn*, *OvIn* e *ZIn*, e, por referência, as variáveis correspondentes a *CyOut*, *OvOut*, *ZOut* e *yOut*. *(Ver Figura 25)*

```
//Bloco da ALU
void ALU(boolean S_0, boolean S_1, boolean S_2, boolean S_3, byte A, byte B, boolean CyIn, boolean OvIn, boolean ZIn, boolean *CyOut, boolean *OvOut, boolean *ZOut, byte *yOut)
{
  switch(S_3 << 3 | S_2 << 2| S_1 << 1 | S_0)
  {
    case B0101:
      //Comando CPLF
      *CyOut = !CyIn;
      *OvOut = !OvIn;
      *ZOut  = !ZIn;
      break;

    case B0110:
      //Comando NOT
      *yOut = ~A;
      *ZOut = !(bitRead(A, 4) << 4 | bitRead(A, 3) << 3 | bitRead(A, 2) << 2 | bitRead(A, 1) << 1 | bitRead(A, 0));
      break;

    case B0111:
      //Comando AND
      *yOut = A & B;
      *ZOut = !(bitRead(A, 4) << 4 | bitRead(A, 3) << 3 | bitRead(A, 2) << 2 | bitRead(A, 1) << 1 | bitRead(A, 0));
      break;

    case B1000:
      //Comando OR
      *yOut = A | B;
      *ZOut = !(bitRead(A, 4) << 4 | bitRead(A, 3) << 3 | bitRead(A, 2) << 2 | bitRead(A, 1) << 1 | bitRead(A, 0));
      break;

    case B1001:
      //Comando ADDC
      *yOut = A + B + CyIn;
      *CyOut = bitRead(A, 5);
      *OvOut = bitRead(A, 4) ^ bitRead(B, 4) ^ bitRead(*yOut, 4) ^ bitRead(A, 5);
      *ZOut = !(bitRead(A, 4) << 4 | bitRead(A, 3) << 3 | bitRead(A, 2) << 2 | bitRead(A, 1) << 1 | bitRead(A, 0));
      break;

    case B1010:
      //Comando SUBB
      *yOut = A - B - CyIn;
      *CyOut = bitRead(A, 5);
      *OvOut = bitRead(A, 4) ^ bitRead(B, 4) ^ bitRead(*yOut, 4) ^ bitRead(A, 5);
      *ZOut = !(bitRead(A, 4) << 4 | bitRead(A, 3) << 3 | bitRead(A, 2) << 2 | bitRead(A, 1) << 1 | bitRead(A, 0));
      break;

    default:
      break;
  }
}
```

Figura 25 - Código ALU

É também implementado o tri-state, *TriState*, que grava o que está no registo A, na Memória de Dados. Confirma-se que o sinal *RD*, está ativo, ou seja, a zero, e, nesse caso, procede a escrita. *(Ver Figura 26)*

```
void TriState()
{
  if(!WR)
  {
    RAM[Y5] = QA;
  }
}
```

Figura 26 - Código TriState

Para simular instruções e valores guardados, foram criadas duas funções que preenchem a *ROM* e a *RAM*, com auxílio a função *random*. *(Ver Figuras 27 e 28)*

```
void PreencherROM()
{
  for(int index = 0; index < 128; index++)
  {
    ROM[index] = random(0, 128);
  }
}
```

Figura 27 - Código Preencher ROM

```
void PreencherRAM()
{
  for(int index = 0; index < 32; index++)
  {
    RAM[index] = random(0, 32);
  }
}
```

Figura 28 - Código Preencher RAM

Finalmente, chegamos ao correr do programa. Na função *setup* iniciamos a comunicação com o arduíno, via o método *Serial.begin()*, definimos o pino do botão que servirá de CLK como *INPUT_PULLUP* no método *pinMode*. Chamamos as funções de preenchimento da Memória de Código e da Memória de Dados, adicionamos o interrupt e ligamo-la. *(Ver Figura 29)*

```
void setup()
{
  Serial.begin(115200);
  pinMode(pinCLK, INPUT_PULLUP);

  PreencherROM();
  PreencherRAM();

  attachInterrupt(pinCLK, MCLK, RISING);
  interrupts();
}
```

Figura 29 - Código Setup

Chegando ao *loop*, invocamos toda a lógica para as funções combinatórias. Ou seja, tudo menos os registos. Começamos por afetar os sinais, De seguida, focamo-nos no circuito combinatório antes do Registo *PC*. Passamos para o Bloco de Registos, depois para a ALU e, de seguida, no circuito combinatório do Registo *A*. Por fim, damos uso ao tri-state. Também chamamos o método "InputUtilizador" mas este será explicado previamente. *(Ver Figura 30)*

```
void loop()
{
  AfetarSinais();
  MUX_2x1((JC & QCy) | (JNZ & !QZ) | (JOv & QOv), 1, bitRead(codigo, 7) << 6 | bitRead(codigo, 7) << 5 | bitRead(codigo, 6) << 4 | bitRead(codigo, 5) << 3 | bitRead(codigo, 4) << 2 | bitRead(codigo, 3) << 1 | bitRead(codigo, 2), &Y1);
  Y2 = Y1 + QPC;
  MUX_2x1(JMP, Y2, bitRead(codigo, 8) << 6 | bitRead(codigo, 7) << 5 | bitRead(codigo, 6) << 4 | bitRead(codigo, 5) << 3 | bitRead(codigo, 4) << 2 | bitRead(codigo, 3) << 1 | bitRead(codigo, 2), &Y3);
  MUX_2x1(SRn, bitRead(codigo, 6) << 4 | bitRead(codigo, 5) << 3 | bitRead(codigo, 4) << 2 | bitRead(codigo, 3) << 1 | bitRead(codigo, 2), QA, &Y4);
  BlocoDeRegistos();
  ALU(bitRead(codigo, 0), bitRead(codigo, 1), bitRead(codigo, 8), bitRead(codigo, 9), Y6, Y5, QCy, QOv, QZ, &DCy, &DOv, &DZ, &Y7);
  MUX_4x1(S0A, S1A, Y5, Y7, RAM[Y5], 0, &Y6);
  TriState();

  InputUtilizador();
}
```

Figura 30 - Código Loop

De forma a analisarmos os resultados, foi necessário criar métodos de *Debug*. O utilizador pode interagir com a consola, permitindo observar cada componente deste microprocessador. O método referenciado anteriormente serve exactamente para isto. Dependendo do input recebido, imprimimos os valores das variáveis desejadas. *(Ver Figura 31)*

```
void InputUtilizador()
{
  if(!displayed)
  {
    Serial.println("Ver conteúdo de: ");
    Serial.println("(r) Registos | (s) Sinais do Módulo de Controlo | (f) Flags | (c) Memória de Código | (d) Memória de Dados");
    displayed = true;
  }

  if(Serial.available())
  {
    int character = Serial.read();
    boolean toBeDisplayed = true;

    switch(character)
    {
      case 'r':
        VerConteudoRegistos();
        break;
      case 's':
        VerSinaisModuloControlo();
        break;
      case 'f':
        VerFlags();
        break;
      case 'd':
        VerMemoriaDeDados();
        break;
      default:
        toBeDisplayed = false;
        break;
    }

    if(toBeDisplayed)
    {
      displayed = false;
    }
  }
}
```

Figura 31 - Código Input Utilizador

Com estes métodos podemos analisar o Conteúdo de Registos *(Ver Figura 32)*, os Sinais de Saída do Módulo de Controlo *(Ver Figura 33)*, as *flags (Ver Figura 34)*, o Conteúdo da Memória de Código *(Ver Figura 35)* e da Memória de Dados *(Ver Figura 36)*.

```
void VerConteudoRegistos()
{
  byte registo0 = bitRead(QRn[0], 4) << 4 | bitRead(QRn[0], 3) << 3 | bitRead(QRn[0], 2) << 2 | bitRead(QRn[0], 1) << 1 | bitRead(QRn[0], 0);
  byte registo1 = bitRead(QRn[1], 4) << 4 | bitRead(QRn[1], 3) << 3 | bitRead(QRn[1], 2) << 2 | bitRead(QRn[1], 1) << 1 | bitRead(QRn[1], 0);

  Serial.println();
  Serial.println("Registos");
  Serial.print("R0 : ");
  Serial.print(registo0, BIN);
  Serial.print(" | R1: ");
  Serial.println(registo1, BIN);
  Serial.println();
  Serial.println();
}
```

Figura 32 - Código Ver Conteúdo de Registos

```
void VerSinaisModuloControlo()
{
  Serial.println();
  Serial.println("Sinais");
  Serial.println("|   EnR   |   EnA   |   RD   |   WR   |   S0A   |   S1A   |   SRn   |   EnCy   |   EnOv   |   EnZ   |   JC   |   JNZ   |   JOv   |   JMP   |");
  Serial.print("|    ");
  Serial.print(EnRn);
  Serial.print("    |    ");
  Serial.print(EnA);
  Serial.print("    |   ");
  Serial.print(RD);
  Serial.print("    |   ");
  Serial.print(WR);
  Serial.print("    |    ");
  Serial.print(S0A);
  Serial.print("    |    ");
  Serial.print(S1A);
  Serial.print("    |    ");
  Serial.print(SRn);
  Serial.print("    |     ");
  Serial.print(EnCy);
  Serial.print("    |     ");
  Serial.print(EnOv);
  Serial.print("    |    ");
  Serial.print(EnZ);
  Serial.print("    |   ");
  Serial.print(JC);
  Serial.print("    |    ");
  Serial.print(JNZ);
  Serial.print("    |    ");
  Serial.print(JOv);
  Serial.print("    |    ");
  Serial.print(JMP);
  Serial.println("    |");
  Serial.println();
  Serial.println();
}
```

Figura 33 - Código Ver Sinais de Saída do Módulo de Controlo

```
void VerFlags()
{
  Serial.println();
  Serial.println("Flags");
  Serial.print("Cy : ");
  Serial.print(QCy);
  Serial.print(" | Ov : ");
  Serial.print(QOv);
  Serial.print(" | Z : ");
  Serial.println(QZ);
  Serial.println();
  Serial.println();
}
```

Figura 34 - Código Ver Flags

```
void VerMemoriaDeCodigo()
{
  Serial.println();
  Serial.println("                                                    Memória de Código                                                    ");
  Serial.println("|---|   0   |   1   |   2   |   3   |   4   |   5   |   6   |   7   |   8   |   9   |   A   |   B   |   C   |   D   |   E   |   F   |");

  for(int line = 0; line < 8; line++)
  {
    Serial.print("| ");
    Serial.print(line);
    Serial.print(" |");

    for(int index = line * 16; index < (line + 1) * 16 ; index++)
    {
      String valorROM = String(ROM[index], 16);

      int valorROMLength = valorROM.length();

      Serial.print("   ");
      for(int bitIndex = 0; bitIndex < 2 - valorROMLength; bitIndex++)
      {
        Serial.print("0");
      }
      Serial.print(ROM[index], HEX);
      Serial.print("   |");
    }
    Serial.println();
  }
  Serial.println();
}
```

Figura 35 - Código Ver Memória de Código

```
void VerMemoriaDeDados()
{
  Serial.println();
  Serial.println("                                                                Memória de Dados                                                                ");
  Serial.println("|---|   0   |   1   |   2   |   3   |   4   |   5   |   6   |   7   |   8   |   9   |   A   |   B   |   C   |   D   |   E   |   F   |");

  for(int line = 0; line < 2; line++)
  {
    Serial.print("| ");
    Serial.print(line);
    Serial.print(" |");

    for(int index = line * 16; index < (line + 1) * 16 ; index++)
    {
      String valorRAM = String(RAM[index], 16);

      int valorRAMLength = valorRAM.length();

      Serial.print("   ");
      for(int bitIndex = 0; bitIndex < 2 - valorRAMLength; bitIndex++)
      {
        Serial.print("0");
      }
      Serial.print(RAM[index], HEX);
      Serial.print("   |");
    }
    Serial.println();
  }
  Serial.println();
}
```

Figura 36 - Código Ver Memória de Dados

3. Programas

3.1. Programa 1

Com este programa, pretendemos somar 127 com 1, de forma a obtermos *Cy*. De seguida, testamos o *JNZ*, que falhará dado que o resultado é 0. Após isto, fazemos *JC* para saltarmos para duas instruções abaixo. Nesta, saltamos continuamente para a instrução 15, pelo *JMP*. *(Ver Figura 37)*

```
void Programa1()
{
  /* Com este programa, pretendemos somar 127 com 1, de forma a obtermos Cy.
     De seguida, testamos o JNZ, que falhará dado que o resultado é 0.
     Após isto, fazemos JC para saltarmos para duas instruções abaixo.
     Nesta, saltamos continuamente para a instrução 15, pelo JMP. */
  ROM[0] = 0b0001111100; //MOV R0, 127
  ROM[1] = 0b0000000001; //MOV A, R0
  ROM[2] = 0b0000000000; //MOV R0, 0
  ROM[3] = 0b0100000000; //MOV @R0, A

  ROM[4] = 0b0010000100; //MOV R1, 1
  ROM[5] = 0b0010000001; //MOV A, R1
  ROM[6] = 011000000000; //MOV @R1, A

  ROM[7] = 0b0000000001; //MOV A, R0
  ROM[8] = 0b0000000011; //MOV A, @R0
  ROM[9] = 0b0000000010; //MOV R0, A

  ROM[10] = 0b0010000011; //MOV A, @R1
  ROM[11] = 0b1000000001; //ADDC A, R0

  ROM[12] = 0b1100001101; //JNZ, 3
  ROM[13] = 0b1100001000; //JC, 2
  ROM[14] = 0b0000000000; //MOV R0, 0
  ROM[15] = 0b1000111111; //JMP, 15
}
```

Figura 37 - Código Programa 1

4. Código

```
//Definir botão para o CLK
#define pinCLK 0

//Memória de Código
word ROM[128];

//Memória de Dados
byte RAM[32];

//Definições da ROM_MC
const int ROM_MC[16] = {0x2C00, 0x1C00, 0x2C80, 0x1500, 0x0800,
0x0C70, 0x1E10, 0x1E10, 0x1E10, 0x1E70, 0x0C01, 0x1E70, 0x0C08,
0x0C04, 0x0C02, 0x0C01};

//Entradas e Saídas dos Registos
byte DPC;               //Entrada Registo PC
volatile byte QPC;      //Saída Registo PC
byte DRn[2];            //Entrada Registo Rn
volatile byte QRn[2];   //Saída Registo Rn
byte DA;                //Entrada Registo A
volatile byte QA;       //Saída Registo A
boolean DCy;            //Entrada Flag Carry
volatile boolean QCy;   //Saída Flag Carry
boolean DOv;            //Entrada Flag Overflow
volatile boolean QOv;   //Saída Flag Overflow
boolean DZ;             //Entrada Flag Zero
volatile boolean QZ;    //Saída Flag Zero

//Saídas do Módulo de Controlo
boolean EnRn, EnA, RD, WR, S0A, S1A, SRn, EnCy, EnOv, EnZ, JC,
JNZ, JOv, JMP;

//Variáveis de Saída
byte Y1, Y2, Y3, Y4, Y5, Y6, Y7;

//Código
word codigo;

//Debug
boolean displayed;

//MUX_2_1
void MUX_2x1(boolean sel, byte in_0, byte in_1, byte *y)
{
  *y = sel ? in_1 : in_0;
}

//MUX_4_1
void MUX_4x1(boolean sel_0, boolean sel_1, byte in_0, byte in_1,
byte in_2, byte in_3, byte *y)
{
  switch(sel_1 << 1 | sel_0)
  {
```

```
    case B00:
        *y = in_0;

      break;

    case B01:
        *y = in_1;

      break;

    case B10:
        *y = in_2;

      break;

    case B11:
        *y = in_3;

      break;
  }
}

//DMUX_1_2
byte DMUX_1x2(boolean sel, boolean D)
{
  return D << sel;
}

//Flip Flop D
void flip_flop_D_Enable(boolean enable, boolean D, boolean *Q)
{
  if(enable)
  {
    *Q = D;
  }
}

void registoComEnable(boolean enable, byte D, byte *Q)
{
  if(enable)
  {
    *Q =  D;
  }
}

//Função MCLK (Ascendente)
void MCLK()
{
  //Atualizar o registo PC
  registoComEnable(1, Y3, &QPC);

  attachInterrupt(pinCLK, MCLKNeg, FALLING);

  Serial.println("Registou PC");
```

```
}

//Função MCLK Negado (Descendente)
void MCLKNeg()
{
  //Atualizar os registos Rn e A, assim como as flags Cy, Ov e Z
  registoComEnable(EnRn, Y4, &QRn[bitRead(codigo, 7)]);
  registoComEnable(EnA, Y6, &QA);
  flip_flop_D_Enable(EnCy, DCy, &QCy);
  flip_flop_D_Enable(EnOv, DOv, &QOv);
  flip_flop_D_Enable(EnZ, DZ, &QZ);

  attachInterrupt(pinCLK, MCLK, RISING);

  Serial.println("Registou OUTROS");
}

void AfetarSinais()
{
  codigo = ROM[QPC];
  byte bitsDeInstrucao = bitRead(codigo, 9) << 3 |
bitRead(codigo, 8) << 2 | bitRead(codigo, 1) << 1 |
bitRead(codigo, 0);
  int sinais = ROM_MC[bitsDeInstrucao];

  JMP = bitRead(sinais, 0);
  JOv = bitRead(sinais, 1);
  JNZ = bitRead(sinais, 2);
  JC = bitRead(sinais, 3);
  EnZ = bitRead(sinais, 4);
  EnOv = bitRead(sinais, 5);
  EnCy = bitRead(sinais, 6);
  SRn = bitRead(sinais, 7);
  S1A = bitRead(sinais, 8);
  S0A = bitRead(sinais, 9);
  WR = bitRead(sinais, 10);
  RD = bitRead(sinais, 11);
  EnA = bitRead(sinais, 12);
  EnRn = bitRead(sinais, 13);
}

//Bloco de Registos
void BlocoDeRegistos()
{
  boolean sel = bitRead(codigo, 7);
  byte enableRegistos = DMUX_1x2(sel, EnRn);
  registoComEnable(bitRead(enableRegistos, 0), Y5, &DRn[0]);
  registoComEnable(bitRead(enableRegistos, 1), Y5, &DRn[1]);
  MUX_2x1(sel, QRn[0], QRn[1], &Y5);
}

//Bloco da ALU
```

```
void ALU(boolean S_0, boolean S_1, boolean S_2, boolean S_3,
byte A, byte B, boolean CyIn, boolean OvIn, boolean ZIn, boolean
*CyOut, boolean *OvOut, boolean *ZOut, byte *yOut)
{
  switch(S_3 << 3 | S_2 << 2| S_1 << 1 | S_0)
  {
    case B0101:
      //Comando CPLF
      *CyOut = !CyIn;
      *OvOut = !OvIn;
      *ZOut  = !ZIn;
      break;

    case B0110:
      //Comando NOT
      *yOut = ~A;
      *ZOut = !(bitRead(A, 4) << 4 | bitRead(A, 3) << 3 |
bitRead(A, 2) << 2 | bitRead(A, 1) << 1 | bitRead(A, 0));
      break;

    case B0111:
      //Comando AND
      *yOut = A & B;
      *ZOut = !(bitRead(A, 4) << 4 | bitRead(A, 3) << 3 |
bitRead(A, 2) << 2 | bitRead(A, 1) << 1 | bitRead(A, 0));
      break;

    case B1000:
      //Comando OR
      *yOut = A | B;
      *ZOut = !(bitRead(A, 4) << 4 | bitRead(A, 3) << 3 |
bitRead(A, 2) << 2 | bitRead(A, 1) << 1 | bitRead(A, 0));
      break;

    case B1001:
      //Comando ADDC
      *yOut = A + B + CyIn;
      *CyOut = bitRead(A, 5);
      *OvOut = bitRead(A, 4) ^ bitRead(B, 4) ^ bitRead(*yOut, 4)
^ bitRead(A, 5);
      *ZOut = !(bitRead(A, 4) << 4 | bitRead(A, 3) << 3 |
bitRead(A, 2) << 2 | bitRead(A, 1) << 1 | bitRead(A, 0));
      break;

    case B1010:
      //Comando SUBB
      *yOut = A - B - CyIn;
      *CyOut = bitRead(A, 5);
      *OvOut = bitRead(A, 4) ^ bitRead(B, 4) ^ bitRead(*yOut, 4)
^ bitRead(A, 5);
      *ZOut = !(bitRead(A, 4) << 4 | bitRead(A, 3) << 3 |
bitRead(A, 2) << 2 | bitRead(A, 1) << 1 | bitRead(A, 0));
      break;
```

```
    default:
      break;
  }
}

void TriState()
{
  if(!WR)
  {
    RAM[Y5] = QA;
  }
}

void PreencherROM()
{
  for(int index = 0; index < 128; index++)
  {
    ROM[index] = random(0, 128);
  }
}

void PreencherRAM()
{
  for(int index = 0; index < 32; index++)
  {
    RAM[index] = random(0, 32);
  }
}

void setup()
{
  Serial.begin(115200);
  pinMode(pinCLK, INPUT_PULLUP);

  PreencherROM();
  PreencherRAM();

  Programa1();

  attachInterrupt(pinCLK, MCLK, RISING);
  interrupts();
}

void loop()
{
  AfetarSinais();
  MUX_2x1((JC & QCy) | (JNZ & !QZ) | (JOv | QOv), 1,
bitRead(codigo, 7) << 6 | bitRead(codigo, 7) << 5 |
bitRead(codigo, 6) << 4 | bitRead(codigo, 5) << 3 |
bitRead(codigo, 4) << 2 | bitRead(codigo, 3) << 1 |
bitRead(codigo, 2), &Y1);
  Y2 = Y1 + QPC;
  MUX_2x1(JMP, Y2, bitRead(codigo, 8) << 6 | bitRead(codigo, 7)
<< 5 | bitRead(codigo, 6) << 4 | bitRead(codigo, 5) << 3 |
```

```
bitRead(codigo, 4) << 2 | bitRead(codigo, 3) << 1 |
bitRead(codigo, 2), &Y3);
  MUX_2x1(SRn, bitRead(codigo, 6) << 4 | bitRead(codigo, 5) << 3
| bitRead(codigo, 4) << 2 | bitRead(codigo, 3) << 1 |
bitRead(codigo, 2), QA, &Y4);
  BlocoDeRegistos();
  ALU(bitRead(codigo, 0), bitRead(codigo, 1), bitRead(codigo,
8), bitRead(codigo, 9), Y6, Y5, QCy, QOv, QZ, &DCy, &DOv, &DZ,
&Y7);
  MUX_4x1(S0A, S1A, Y5, Y7, RAM[Y5], 0, &Y6);
  TriState();

  InputUtilizador();
}

//Métodos de Debug
void VerConteudoRegistos()
{
  byte registo0 = bitRead(QRn[0], 4) << 4 | bitRead(QRn[0], 3)
<< 3 | bitRead(QRn[0], 2) << 2 | bitRead(QRn[0], 1) << 1 |
bitRead(QRn[0], 0);
  byte registo1 = bitRead(QRn[1], 4) << 4 | bitRead(QRn[1], 3)
<< 3 | bitRead(QRn[1], 2) << 2 | bitRead(QRn[1], 1) << 1 |
bitRead(QRn[1], 0);

  Serial.println();
  Serial.println("Registos");
  Serial.print("R0 : ");
  Serial.print(registo0, BIN);
  Serial.print(" | R1: ");
  Serial.println(registo1, BIN);
  Serial.println();
  Serial.println();
}

void VerSinaisModuloControlo()
{
  Serial.println();
  Serial.println("Sinais");
  Serial.println("|   EnR   |   EnA   |   RD   |   WR   |   S0A
|   S1A   |   SRn   |   EnCy   |   EnOv   |   EnZ   |   JC   |
JNZ   |   JOv   |   JMP   |");
  Serial.print("|    ");
  Serial.print(EnRn);
  Serial.print("    |    ");
  Serial.print(EnA);
  Serial.print("    |   ");
  Serial.print(RD);
  Serial.print("    |   ");
  Serial.print(WR);
  Serial.print("    |    ");
  Serial.print(S0A);
  Serial.print("    |    ");
  Serial.print(S1A);
```

```
  Serial.print("    |    ");
  Serial.print(SRn);
  Serial.print("    |     ");
  Serial.print(EnCy);
  Serial.print("    |     ");
  Serial.print(EnOv);
  Serial.print("    |    ");
  Serial.print(EnZ);
  Serial.print("    |   ");
  Serial.print(JC);
  Serial.print("    |    ");
  Serial.print(JNZ);
  Serial.print("    |    ");
  Serial.print(JOv);
  Serial.print("    |    ");
  Serial.print(JMP);
  Serial.println("    |");
  Serial.println();
  Serial.println();
}

void VerFlags()
{
  Serial.println();
  Serial.println("Flags");
  Serial.print("Cy : ");
  Serial.print(QCy);
  Serial.print(" | Ov : ");
  Serial.print(QOv);
  Serial.print(" | Z : ");
  Serial.println(QZ);
  Serial.println();
  Serial.println();
}

void VerMemoriaDeCodigo()
{
  Serial.println();
  Serial.println("
Memória de Código
");
  Serial.println("|---|   0    |   1    |   2    |   3    |   4
|   5    |   6    |   7    |   8    |   9    |   A    |   B    |
C    |   D    |   E    |   F    |");

  for(int line = 0; line < 8; line++)
  {
    Serial.print("| ");
    Serial.print(line);
    Serial.print(" |");

    for(int index = line * 16; index < (line + 1) * 16 ;
index++)
    {
```

```
      String valorROM = String(ROM[index], 16);

      int valorROMLength = valorROM.length();

      Serial.print("   ");
      for(int bitIndex = 0; bitIndex < 2 - valorROMLength;
bitIndex++)
      {
        Serial.print("0");
      }
      Serial.print(ROM[index], HEX);
      Serial.print("   |");
    }
    Serial.println();
  }
  Serial.println();
}

void VerMemoriaDeDados()
{
  Serial.println();
  Serial.println("
Memória de Dados
");
  Serial.println("|---|   0    |   1    |   2    |   3    |   4
|   5    |   6    |   7    |   8    |   9    |   A    |   B    |
C    |   D    |   E    |   F    |");

  for(int line = 0; line < 2; line++)
  {
    Serial.print("| ");
    Serial.print(line);
    Serial.print(" |");

    for(int index = line * 16; index < (line + 1) * 16 ;
index++)
    {
      String valorRAM = String(RAM[index], 16);

      int valorRAMLength = valorRAM.length();

      Serial.print("   ");
      for(int bitIndex = 0; bitIndex < 2 - valorRAMLength;
bitIndex++)
      {
        Serial.print("0");
      }
      Serial.print(RAM[index], HEX);
      Serial.print("   |");
    }
    Serial.println();
  }
  Serial.println();
}
```

```
void InputUtilizador()
{
  if(!displayed)
  {
    Serial.println("Ver conteúdo de: ");
    Serial.println("(r) Registos | (s) Sinais do Módulo de
Controlo | (f) Flags | (c) Memória de Código | (d) Memória de
Dados");
    displayed = true;
  }

  if(Serial.available())
  {
    int character = Serial.read();
    boolean toBeDisplayed = true;

    switch(character)
    {
      case 'r':
        VerConteudoRegistos();
        break;
      case 's':
        VerSinaisModuloControlo();
        break;
      case 'f':
        VerFlags();
        break;
      case 'c':
        VerMemoriaDeCodigo();
        break;
      case 'd':
        VerMemoriaDeDados();
        break;
      default:
        toBeDisplayed = false;
        break;
    }

    if(toBeDisplayed)
    {
      displayed = false;
    }
  }
}

void Programa1()
{
  /* Com este programa, pretendemos somar 127 com 1, de forma a
obtermos Cy.
     De seguida, testamos o JNZ, que falhará dado que o
resultado é 0.
     Após isto, fazemos JC para saltarmos para duas instruções
abaixo.
```

```
    Nesta, saltamos continuamente para a instrução 15, pelo
JMP. */
  ROM[0] = 0b0001111100; //MOV R0, 127
  ROM[1] = 0b0000000001; //MOV A, R0
  ROM[2] = 0b0000000000; //MOV R0, 0
  ROM[3] = 0b0100000000; //MOV @R0, A

  ROM[4] = 0b0010000100; //MOV R1, 1
  ROM[5] = 0b0010000001; //MOV A, R1
  ROM[6] = 011000000000; //MOV @R1, A

  ROM[7] = 0b0000000001; //MOV A, R0
  ROM[8] = 0b0000000011; //MOV A, @R0
  ROM[9] = 0b0000000010; //MOV R0, A

  ROM[10] = 0b0010000011; //MOV A, @R1
  ROM[11] = 0b1000000001; //ADDC A, R0

  ROM[12] = 0b1100001101; //JNZ, 3
  ROM[13] = 0b1100001000; //JC, 2
  ROM[14] = 0b0000000000; //MOV R0, 0
  ROM[15] = 0b1000111111; //JMP, 15
}
```